Motion Prediction for Autonomous Driving with CNN, LSTM, and STL

Khushal Brahmbhatt
Oregon State University
Corvallis, OR
brahmbhk@oregonstate.edu

Nicole Fronda
Oregon State University
Corvallis, OR
frondan@oregonstate.edu

Abstract

Vehicles with autonomous and semi-autonomous capabilities are becoming increasingly prevalent. A key component of the software stack for autonomous driving is motion prediction. In this paper, we implement and evaluate different deep learning architectures as solutions for vehicle trajectory prediction. We use the Lyft Prediction dataset to train and evaluate these architectures, which include CNN architectures such as ResNet50 and MobileNetV2, as well as combined CNN and LSTM architectures. We also present a novel method for extracting logical representations of the predicted trajectories using an RNN of custom-built cells which enables learning of Signal Temporal Logic (STL) formulae.

1. Introduction

Engineering robust and accurate motion planning solutions for autonomous driving on urban roads is central to making self-driving cars a reality. Although it remains unclear just how much self-driving cars are likely to improve road safety by [4], recent test data from key industry leaders [11] suggests they will have a significant impact in reducing the number of road accidents and fatalities per year.

An important part of motion planning in urban driving scenarios is motion prediction for surrounding vehicles. Anticipating trajectories of surrounding vehicles enables collision-free planning. This paper will implement and evaluate a selection of state-of-the-art deep learning models for motion prediction.

Equally important to motion prediction is confidence in the correctness of the prediction. Signal Temporal Logic (STL) is a formalism that can be used to describe and reason about continuous time signals, such as vehicle trajectories, and allows for quantification of the robustness of signals produced by a model. In this paper, we present a novel method for learning STL formulae using an RNN architecture with custom cells. We use this method to learn an STL representation of our deep learning models' trajectory predictions, and evaluate robustness of real trajectories with respect to the learned STL formula.

Our main contributions in this paper are the following:

1. Evaluation of different deep learning architectures for motion prediction through experimentation using the Lyft Prediction dataset.
2. A novel method for learning STL formulae using deep learning.
3. An application of this method to describing trajectories produced by a motion prediction model from 1).

The rest of the paper is organized as follows: Section 2 goes over the preliminaries of STL. Section 3 reviews previous work in motion prediction and STL learning. Section 4 outlines our implementations of different deep learning motion prediction models and our experimental setup for evaluating these models. This section also outlines our methodology for learning STL formulae. Our experimental results are presented and discussed in Section 5. Finally, we present our conclusions and future scope in Section 6.

2. Preliminaries

2.1. STL Syntax and Semantics

Signal Temporal Logic (STL) [7] is a logical formal language for describing temporal properties of signals. It can be used for specification of desired system behaviours such as "The vehicle must *always* be centered within a lane" or "The vehicle may *eventually* come to a stop".

In STL, a *signal* given by $\mathbf{x}$ is a function from a time domain to a value domain. The value of a signal at a point in time t is given by $x[t]$ The syntax of STL is defined recursively as:

$$\phi := \mu \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \mathbf{F}_{[a,b]}\phi \mid \mathbf{G}_{[a,b]}\phi$$

where the atomic predicate μ is defined as $\mu := f(\mathbf{x}) \sim c$. Note that $f(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}$, $\sim \in (<, \geq)$, and c is a scalar constant. The temporal operators $\mathbf{F}_{[a,b]}$ and $\mathbf{G}_{[a,b]}$ are read

as "eventually between time a and b" and "always between time a and b", respectively.

The semantics for STL are given as:

$$x[t] \models \mu \iff f(\mathbf{x}) \sim c \quad (1)$$
$$x[t] \models \neg\phi \iff \mathbf{x}[t] \not\models \phi \quad (2)$$
$$x[t] \models \phi_1 \wedge \phi_2 \iff \mathbf{x}[t] \models \phi_1 \wedge \mathbf{x}[t] \models \phi_2 \quad (3)$$
$$x[t] \models \phi_1 \vee \phi_2 \iff \mathbf{x}[t] \models \phi_1 \vee \mathbf{x}[t] \models \phi_2 \quad (4)$$
$$x[t] \models \mathbf{F}_{[a,b]} \iff \exists\tau \in [t+a, t+b] s.t. \mathbf{x}[\tau] \models \phi \quad (5)$$
$$x[t] \models \mathbf{G}_{[a,b]} \iff \forall\tau \in [t+a, t+b] s.t. \mathbf{x}[\tau] \models \phi \quad (6)$$

For this project, we do not consider time bounds for **F** and **G**.

2.2. STL Robustness Semantics

The robust semantics of STL give a quantitative measure of how well a STL formula ϕ is satisfied by a signal $\mathbf{x}$. Robustness $r(\mathbf{x}, \phi)$ of $\mathbf{x}$ can also be described as its "distance to violation" of ϕ. A positive robustness value indicates that the signal satisfies ϕ and a negative robustness value indicates that the signal violates ϕ.

The robustness of STL are calculated recursively as:

$$r(\mathbf{x}, \mu) = f(\mathbf{x}) - c \iff \mu \models f(\mathbf{x}) \geq c \quad (7)$$
$$r(\mathbf{x}, \mu) = c - f(\mathbf{x}) \iff \mu \models f(\mathbf{x}) < c \quad (8)$$
$$r(\mathbf{x}, \neg\phi) = -r(\mathbf{x}, \phi) \quad (9)$$
$$r(\mathbf{x}, \phi_1 \wedge \phi_2) = min(r(\mathbf{x}, \phi_1), \mathbf{x}, \phi_2) \quad (10)$$
$$r(\mathbf{x}, \phi_1 \vee \phi_2) = max(r(\mathbf{x}, \phi_1), \mathbf{x}, \phi_2) \quad (11)$$
$$r(\mathbf{x}, \mathbf{F}\phi) = sup(r(\mathbf{x}, \phi) \quad (12)$$
$$r(\mathbf{x}, \mathbf{G}\phi) = inf(r(\mathbf{x}, \phi) \quad (13)$$

3. Related Work

Recent works have tried to accomplish motion prediction by directly learning from semantic maps using deep learning methods. Houston et al. [2] use a Convolutional Neural Network (CNN) to directly predict trajectories of surrounding vehicles from semantic map images generated from the dataset containing road geometry as well as obstacle masks and their history states. By combining CNNs with Long Short-Term Memory (LSTM) networks, [1] show that history state information of the vehicle such as its speed, acceleration and heading can also be leveraged along with the semantic maps to predict more accurate trajectories. The authors of [12] use a single neural network to jointly perform object detection, motion prediction and motion planning. Most deep learning-based motion prediction methods can also be extended to do motion planning. Thus an in-depth review of deep learning methods for behavior prediction for autonomous driving applications can be found in [9].

While deep learning models are effective at motion prediction, they often lead to unexplainable solutions that are difficult to evaluate during failure cases. Formal methods such as STL provide a toolkit that can be used for interpreting deep learning model outcomes. Learning an STL formula from a deep learning model can provide some explainability of that model.

Previous research in learning STL formulae generally involves iteratively building a new formula given some parameters for the formula structure. Then at each iteration, the formula is evaluated by calculating the robustness of signals in a training dataset with respect to the formula. This is repeated until some threshold for correct classification of the signals is met. Correct classification in this task is correct output of a positive robustness value for a signal that should satisfy the formula, and correct output of a negative robustness value for a signal that should not satisfy the formula. The method proposed by Kong, et al in [5] iteratively builds new formula by organizing smaller formulae in a graph, and then growing and pruning the graph to yield new formulae at each iteration. In the work by Mohammadinejad, et al in [8], all possible formulae are enumerated and stored in database for evaluation. A signature-based optimization is used to detect which formulae in the database are similar, and removes them to reduce the number of formulae for evaluation. Our method for learning STL formulae differs from these previous works in that the formula structure to be learned is embedded in a RNN architecture. We use RNNs as our base architecture due to their efficiency at processing sequential data.

4. Methods

4.1. Dataset

The Lyft Prediction dataset [2] is a self-driving dataset by Lyft for motion prediction and planning tasks. It consists of over 1000 hours of driving data from Palo Alto, California and a semantic and aerial map with over 15,000 labeled elements and hand-annotated road and lane boundaries. The dataset is split into 170,000 scenes of 25 seconds each, and each scene captures the perception output at 10Hz. Our goal is to predict a 5-second future trajectory of surrounding vehicles using this data. The l5kit library [6] provides convenient tools to rasterize the semantic map with the surrounding object masks as shown in Figure 1.

4.2. Vehicle Trajectory Prediction

4.2.1 Baseline

Our goal is to predict a 5-second future trajectory using a 1-second history from the scenes. Since the data is captured at 10Hz, we need to predict the x,y coordinates for the next 50 timesteps. Our prediction output is therefore a vector of length 100.

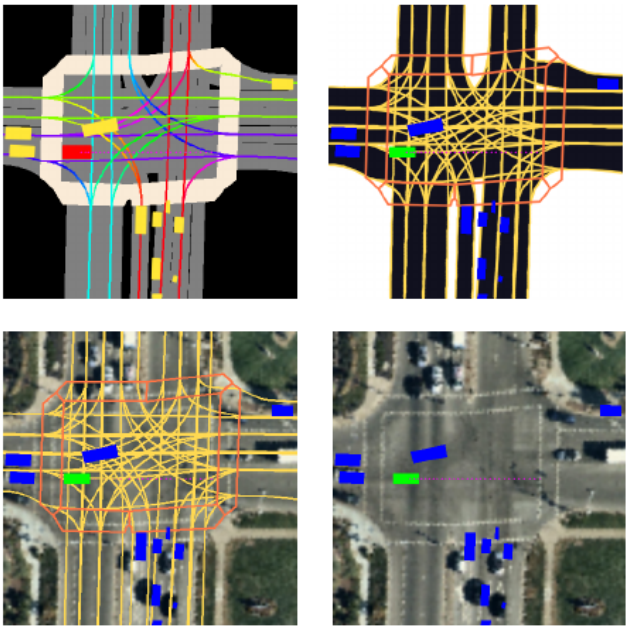


Figure 1: Training images from the dataset obtained using l5kit toolkit [2]

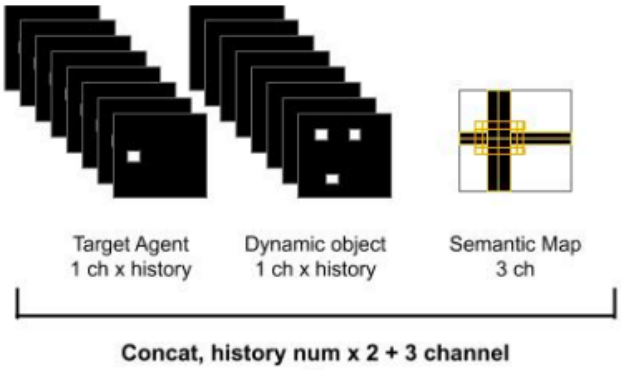


Figure 2: Input channels for the training images [3]

The baseline model uses a simple pre-trained ResNet50 CNN model that works directly on the rasterized semantic map images as inputs and outputs 50 x,y coordinates as the trajectory prediction. The input images are of size 224x224 and consist of 25 channels. We refer to the agent whose trajectory we are predicting as the target agent, and the surrounding moving agents as the dynamic objects. Since we are encoding information from a 1-second history, we have 11 channels (10 history frames + 1 current frame) for the target agent mask, 11 channels for the dynamic object masks, and 3 channels (R, G, B) for the semantic map. This is shown in Figure 2.

4.2.2 CNN + LSTM encoder

The baseline model only captures the agent history positions and heading. In order to leverage the rest of the temporal information of the agent states available in the dataset such as the agent's velocity, acceleration and yaw rate, we combine the features learnt from a base CNN with the temporal state features and encode them using an LSTM as shown in Figure 3.

The features output by a base CNN model are concatenated with state inputs for the previous 10 frames. The concatenated features are then passed through two fully-connected layers of size 4096 and 128. The output of the last fully-connected layer is given as input to the LSTM, and the output of the LSTM is passed through another fully-connected layer to get 50 x,y future coordinate predictions.

4.2.3 CNN + LSTM encoder-decoder

While the CNN + LSTM encoder model is able to capture the temporal information of the agent states, it does not make intuitive sense to combine the state features with the CNN output since the temporal information from the channels of the images passed through the CNN will not be retained in a sequential format that can be passed to an LSTM. We therefore decide to first encode only the agent's states through an LSTM before concatenating them with the CNN output as shown in Figure 4.

The concatenated features are then passed through two fully-connected layers as in the previous model. We then use an LSTM decoder to get the target positions at each step, instead of a fully-connected layer at the encoder output as in the previous model. This is more intuitive since the target positions along the trajectory also have a temporal structure.

4.2.4 Experiments

We trained 5 different models for our experiments: 1) a ResNet50 baseline, 2) ResNet50 + LSTM encoder, 3) ResNet50 + LSTM Encoder-Decoder, 4) MobileNetV2 baseline and 5) MobileNetV2 + LSTM Encoder-Decoder. All our experiments were run on a Tesla P100 GPU on Kaggle. However, CPU turned out to be the major bottleneck for our experiments since the rasterization process in the l5kit library used to convert the semantic map and perception output data to images used CPU operations. This meant we were utilizing less than 20% of the GPU computing power available and could only train over a small section of the dataset. Each experiment took 6-7 hours (limited by Kaggle's maximum runtime) to run and was trained over 300k frames. In comparison, the full dataset available to us was over 22M frames.

We trained all our models using a batch size of 12 and 25k iterations. Initially, we used Adam optimizer with a

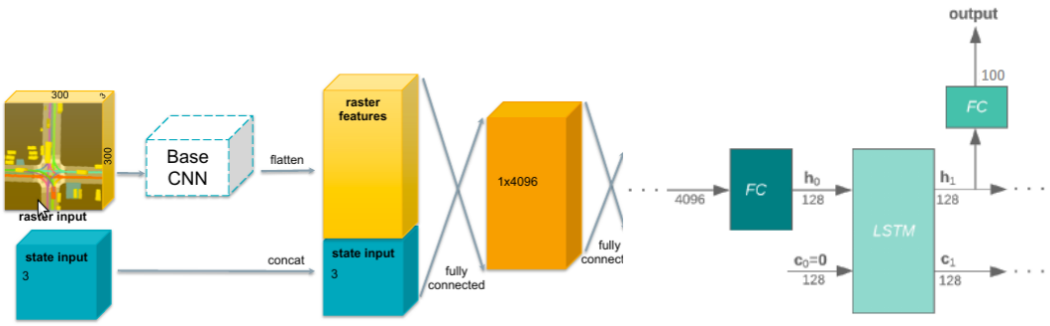


Figure 3: CNN + LSTM encoder model [1]

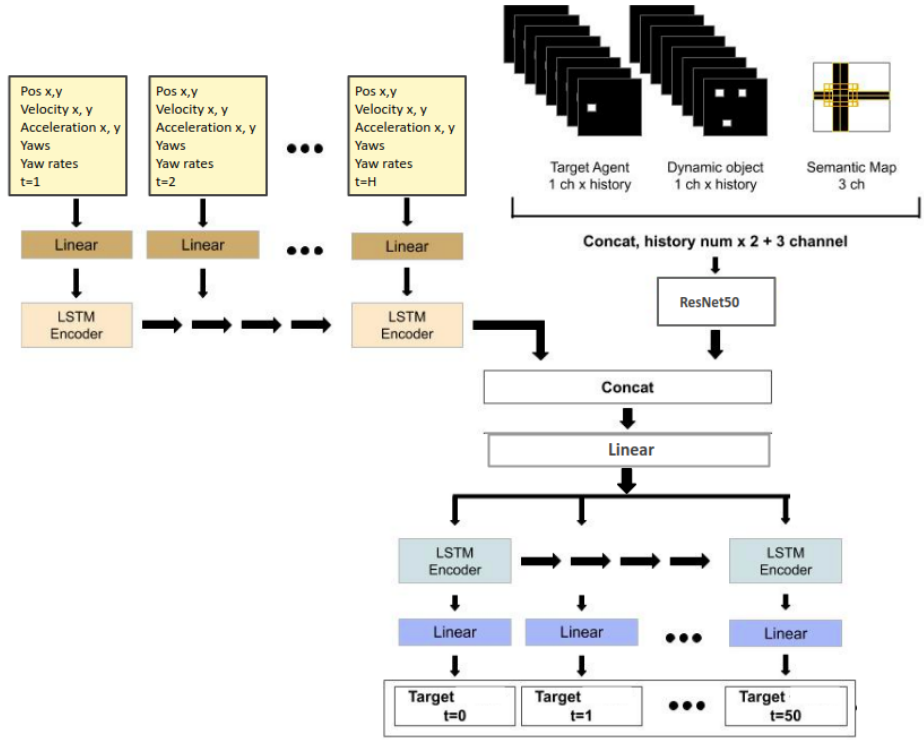


Figure 4: CNN + LSTM encoder-decoder model [3]

learning rate of 1e-3, but this led to gradient explosion as shown in Figure 5. We therefore switched to AdamW optimizer which decouples the weight decay from the gradient update. We also used a cosine annealing learning rate scheduler between learning rates 1e-5 and 1e-8. This led to more stable training, and the results are shown in section 5.

4.2.5 Evaluation

We have used 4 metrics to evaluate our models:

(i) **Multimodal Negative Log-Likelihood (NLL)**

Since we can predict more than one trajectory with different confidences, this metric allows us to calculate the negative log-likelihood over all the predicted trajectories. For k predicted trajectories, we can calculate the NLL as $-\log \sum_k e^{\log(c^k) - \frac{1}{2}\sum_t (\bar{x}_t^k - x_t)^2 + (\bar{y}_t^k - y_t)^2}$.

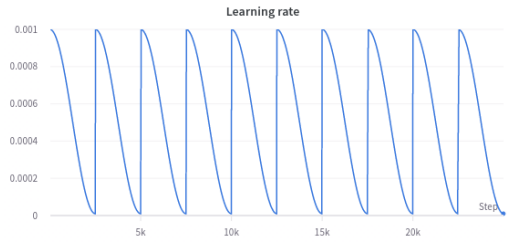
(ii) **Root Mean Squared Error (RMSE)**

This is the square root of the mean squared error across all future trajectory positions. It can be computed from the negative log-likelihood as $\sqrt{\frac{2\times \text{NLL}}{T}}$ where T is the number of target positions.

(iii) **Average Displacement Error**

This is simply the mean of the displacement error between the predicted and ground truth positions along

(a) Loss curve for gradient explosion during learning



(b) Learning rate scheduling using cosine annealing

Figure 5: Gradient explosion and learning rate scheduler

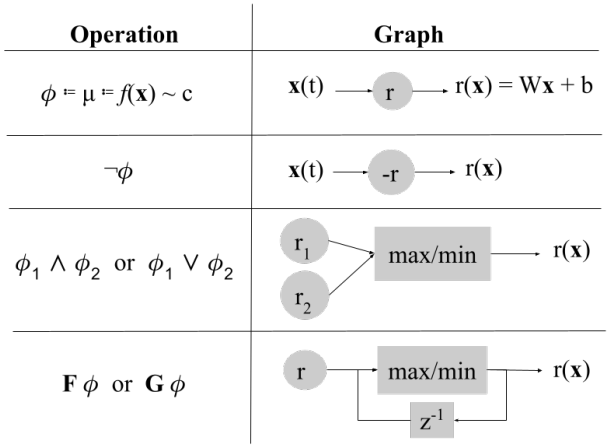


Figure 6: Given the STL robustness semantics described in 2.2, we represent the STL predicate and each operator as a graph structure. These structures are used to construct custom RNN cells.

the trajectory.

(iv) **Final Displacement Error**

This is the displacement error between the last predicted future position along the trajectory and the last target position on the trajectory i.e. when t=50.

4.3. Learning STL Formulae for Model Predictions

Our process for learning an STL formula for trajectory predictions consists of two steps:

1. Train a RNN to distinguish between trajectories produced by our model (one of the stand-alone CNN or CNN+LSTM implementations) and other (random) trajectories.

2. Extract quantized weights from the RNN to recover the structure of an STL formula.

In order to accomplish task 2), the RNN architecture for task 1) must be constructed in a way such that the operations within the RNN layers match the robust semantics of STL predicates and operators. We achieve this by representing each predicate and operator as a graph structure which we then use to develop custom RNN cells. These structures are shown in Figure 6.

We then construct an STL-based RNN with these cells and train the network to classify whether a given trajectory, consisting of a series of (x,y) coordinates of length 50, is generated by one of our vehicle trajectory models or is otherwise a randomly generated trajectory. Trajectories output from a model are given a label of 1. Random trajectories are given a label of -1. The idea is that during training the RNN will learn which combination of STL predicates and operators (cells), will produce positive robustness for the positively labeled trajectories (the model produced trajectories). The "selection" of predicates and operators by the RNN is done through quantization of the weights with a one-hot constraint. That is, for a group of candidate predicates or operators for an STL formula, only one of the quantized weights corresponding to each predicate or operator in the group may equal 1 and the rest must equal 0. An illustration of this selection process is shown in Figure 7. We modify the binary quantization method proposed in [10] to implement this one-hot constraint. After training, all weights are extracted, and the predicates and operators with quantized weights of 1 form the final STL formula.

The architecture of the RNN we used to learn an STL formula for the predicted trajectories is presented in Figure 8. This architecture was developed to learn a formula of the form $\square(\phi_1) \wedge \square(\phi_2)$, where $\square$ denotes either a **F** or **G** operator. Both ϕ_1 and ϕ_2 will have the form $(\mathbf{x} \sim c) \triangle (\mathbf{y} \sim c)$, where $\triangle$ denotes either a $\wedge$ or $\vee$ operator, and **x** and **y** correspond to the (x,y) coordinates of the trajectory. It is important to note that constructing an STL-based architecture requires some domain knowledge of the source dataset to inform a reasonable final formula structure. The formula structure learned by the architecture shown in Figure 8 was arbitrarily selected for this project.

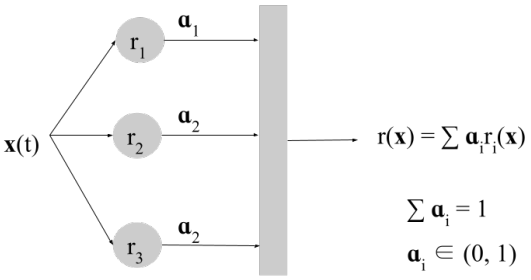


Figure 7: An example of how the STL-based RNN architecture learns which predicate or operator to select for the formula using a one-hot constraint on quantized weights. Here, there are three possible predicates (r_1, r_2, r_3) which belong to a formula. The weights (α_1, α_2, α_3) determine which of these predicates will be selected. Exactly one of these weights will equal 1, and its corresponding predicate will be selected for the formula.

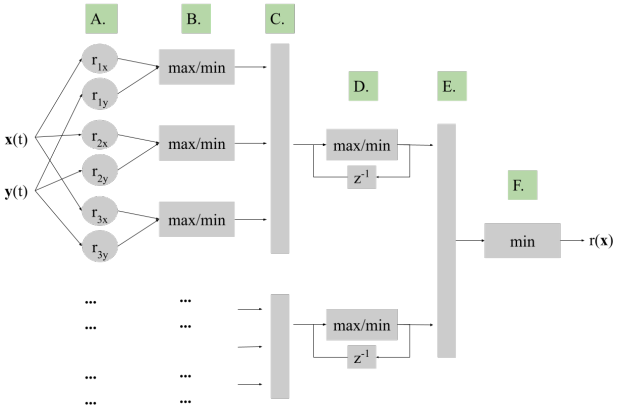


Figure 8: The STL-based RNN architecture used to learn a formula for the model predicted trajectories. Layer A learns atomic predicates for the **x** and **y** coordinates. Layer B learns whether each pair of predicates should form a conjunction ($\wedge$) or disjunction ($\vee$). Layer C learns which conjunction/disjunction belongs in the formula. Layer D learns whether the conjunction/disjunction is nested within an **F** or **G** operator. Layer E concatenates two **F** or **G** operators. Layer F combines the outputs of Layer E in an $\wedge$ operator. This architecture learns a formula of the form $\square(\phi_1) \wedge \square(\phi_2)$

Figure 9: Average training loss of some of the models

Model	NLL	RMSE	ADE (m)	FDE (m)
ResNet50	202.02	1.54	5.38	9.54
ResNet50 + LSTM Encoder-Decoder	233	1.53	5.39	9.53
MobileNetV2	239.95	1.65	5.42	9.59
MobileNetV2 + LSTM Encoder	722.59	9.21	7.6	13.18
MobileNetV2 + LSTM Encoder-Decoder	337.54	1.92	5.49	9.74

Table 1: Comparison of different model performances

5. Results

5.1. Trajectory Predictions

The training loss curves of 3 of the models are shown in Figure 9. Due to the large amount of training time needed to train these models as well as the computational bottleneck on the CPU, we did not run validation along with the training loop. We directly used inference results to evaluate our models using the metrics mentioned above. Our evaluation results are summarized in Table 1.

From our results, we observe that the baseline models using only a CNN are able to outperform the CNN + LSTM models. It is unclear with the current results if given more training data, the recurrent models could perform better than the CNN baselines. We expect the displacement errors to also drop significantly below 1m when trained on the full dataset. However, we can see from some of the trajectory visualizations in Figure 10 that even after being trained on just 1% of the dataset, some of the trajectory predictions are already very close to the ground truths.

5.2. STL Formulae

We used the STL-based RNN architecture in 8 to learn a formula that describes the predicted trajectories of our ResNet50 and MobileNetV2+LSTM model, our best and worst performing models respectively. We also learn a formula on the ground truth trajectories themselves. After learning a formula, we evaluate how representative it is of the ground truth trajectories by calculating the proportion of positive robustness (PPR), or proportion of trajectories which yield a positive robustness value with respect to the formula. The resulting formula and PPR for each of these experiments is presented in Table 2. Additionally, Figure 11

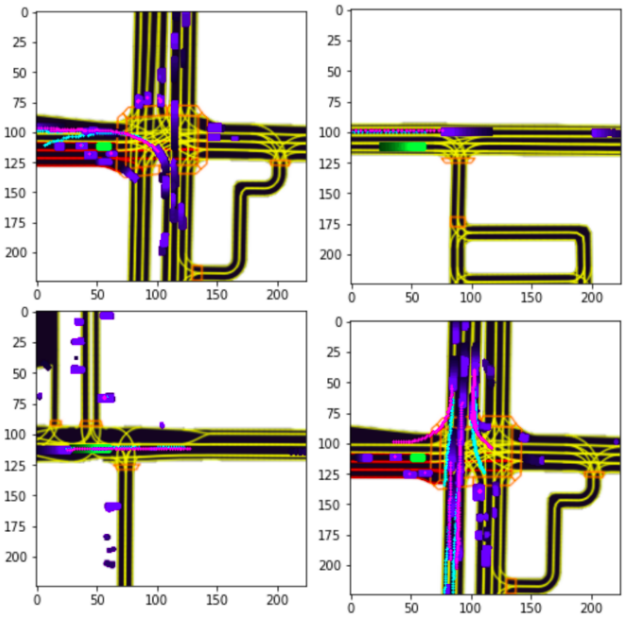


Figure 10: Visualizing trajectory predictions: ground truths are pink, prediction are cyan.

illustrates the bounds for positive robustness for the formulae alongside the ground truth trajectories.

Our results show that more than 99% of the the ground truth trajectories are robust to each of the learned formulae. This means that a formula derived from the model trajectories can almost wholly describe real trajectories. However, we cannot yet say anything of the correctness or robustness of the models themselves, just that trajectories produced by the models behave similarly to real trajectories through the lens of this formula structure. Next steps will be to compare our formulae and robustness metric with state-of-the art STL inference technique such as those discussed in [5] and [8].

6. Conclusion

In this paper, we implemented and evaluated different deep learning model architectures for motion prediction. We also proposed a novel method for learning an STL formula for these predictions using an RNN. Our results show that CNN models alone are able to learn satisfactory trajectory predictions from semantic maps. Temporal state information may not as significantly contribute to trajectory predictions as the lane geometry and surrounding objects. The STL learned formulae also show good coverage of the true trajectories, but appear to specify redundant information. This indicates that a simpler formula structure could be learned.

Future work includes training the discussed model architectures on larger datasets, and comparing our STL-based RNN architecture with traditional methods for learning STL formulae.

References

[1] N. Djuric, V. Radosavljevic, H. Cui, T. Nguyen, F.-C. Chou, T.-H. Lin, N. Singh, and J. Schneider. Uncertainty-aware short-term motion prediction of traffic actors for autonomous driving. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 2095–2104, 2020.

[2] J. Houston, G. Zuidhof, L. Bergamini, Y. Ye, A. Jain, S. Omari, V. Iglovikov, and P. Ondruska. One thousand and one hours: Self-driving motion prediction dataset. *arXiv preprint arXiv:2006.14480*, 2020.

[3] Y. Inoue. 14th place solution. custom mask and lstm encoder/decoder. `https://www.kaggle.com/c/lyft-motion-prediction-autonomous-vehicles/discussion/201143`, 2020.

[4] N. Kalra and S. M. Paddock. Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability? *Transportation Research Part A: Policy and Practice*, 94:182–193, 2016.

[5] Z. Kong, A. Jones, A. Medina Ayala, E. Aydin Gol, and C. Belta. Temporal logic inference for classification and prediction from data. In *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control*, HSCC '14, page 273–282, New York, NY, USA, 2014. Association for Computing Machinery.

[6] Lyft. l5kit. `https://github.com/lyft/l5kit`, 2020.

[7] O. Maler and D. Nickovic. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, volume 3253, pages 152–166, 01 2004.

[8] S. Mohammadinejad, J. V. Deshmukh, A. G. Puranic, M. Vazquez-Chanlatte, and A. Donzé. Interpretable classification of time-series data using efficient enumerative techniques. In *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*, HSCC '20, New York, NY, USA, 2020. Association for Computing Machinery.

[9] S. Mozaffari, O. Y. Al-Jarrah, M. Dianati, P. Jennings, and A. Mouzakitis. Deep learning-based vehicle behavior prediction for autonomous driving applications: A review. *IEEE Transactions on Intelligent Transportation Systems*, 2020.

[10] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016.

[11] M. Schwall, T. Daniel, T. Victor, F. Favaro, and H. Hohnhold. Waymo public road safety performance data. *arXiv preprint arXiv:2011.00038*, 2020.

[12] W. Zeng, S. Wang, R. Liao, Y. Chen, B. Yang, and R. Urtasun. Dsdnet: Deep structured self-driving network. In *European Conference on Computer Vision*, pages 156–172. Springer, 2020.

Source Trajectories	Learned STL Formula	PPR
Ground Truth	$\mathbf{G}((\mathbf{x} \geq 27.88) \vee (\mathbf{y} < 119.80)) \wedge \mathbf{G}((\mathbf{x} < 0.47) \wedge (\mathbf{y} < 1.46))$	% 99.83
ResNet50	$\mathbf{G}((\mathbf{x} \geq 27.94) \vee (\mathbf{y} < 80.00)) \wedge \mathbf{G}((\mathbf{x} < 0.48) \wedge (\mathbf{y} < 1.49))$	% 99.85
MobileNetV2+LSTM	$\mathbf{G}((\mathbf{x} \geq 29.80) \vee (\mathbf{y} < 92.34)) \wedge \mathbf{G}((\mathbf{x} < 0.45) \wedge (\mathbf{y} < 1.33))$	% 99.66

Table 2: The learned STL formulae from ground truth trajectories and model predicted trajectories and their proportion of positive robustness (PPR) against the ground truth trajectories.

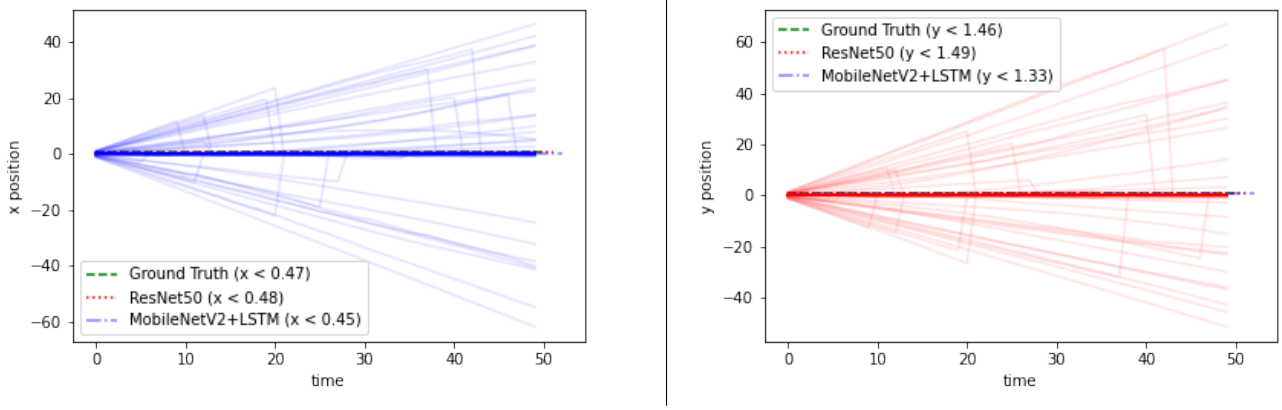


Figure 11: Ground Truth x and y coordinates of a 50 timestep trajectory. The bounds found by each learned STL formulae are also included.